The NYU Ultracomputer

Malvin H. Kalos[1]

Courant Institute, N.Y.U.
251 Mercer Street
New York, N.Y. 10012

Ultracomputer Note #48

April, 1983

ABSTRACT

The architecture of the NYU Ultracomputer is outlined. This is a proposed "MIMD" parallel machine comprising thousands of independently programmable fast processing elements connected to a large shared memory by a switching network. The latter has the geometry of an Omega-network, with switches each having enough memory and processing power to yield a bandwidth proportional to the number of processors. The network, which is pipelined, also supports the parallel execution of the "fetch-and-add" primitive important for process coordination. Questions relating to operating systems, programmability, and efficiency of applications codes are also discussed.

INTRODUCTION

In attempting to foresee the future it is always interesting and sometimes even instructive to review the past. The evolution of modern computers can be traced along two parallel (and interacting) paths. One is the development of basic technology - from mechanical relays to VLSI chips - so as to become cheaper, smaller, and faster. The other is the logical organization of computers into increasingly sophisticated arrangements emphasizing more parallel capability. This development has ranged from parallel addition of all bits of a word, through parallel and autonomous functional units, to the effectively parallel operation of a contemporary pipelined vector processor.

---

[1]Invited paper for the 1983 Mathematics and Computation Topical Meeting of the American Nuclear Society. Supported by the Applied Mathematical Sciences Program of the Department of Energy under grant DE-AC02-76ER03077, and by the National Science Foundation under grant NSF-MC79-07804

One forsees, therefore, the development of parallelism at an ever higher level. But it is not clear which kind of parallelism offers the most promise. There might be four kinds. The first would use many functional units arranged so that the machine as a whole looks to the user like a sequential computer. This would be a very complex logical design and it is not clear that it could be pushed beyond about 200 million instructions per second (mips). A second approach would make vector units ever more powerful and would include autonomous functional units (as in the CRAY) for different operations. The third direction would couple together several (perhaps as many as 16) sophisticated vector processors. The fourth possible line of development lies in the use of very many simple processors tightly coupled so as to cooperate effectively on major computations. The NYU Ultracomputer is of this last type and its conception is predicated upon the ideas that it would be favored by the technical and economic evolution of computers in the near future and that it would potentially be the most easily used, at least with computer languages and compilers available now and in the foreseeable future.

NEAR TERM TECHNOLOGY

We assume the following three developments in basic computer technology between now and 1990: Microcomputers will become more powerful, faster, and probably cheaper as well; thus we assume the existence of 32 bit microprocessors on one or a very few chips, capable of 10-20 mips and costing less than $100. We assume further that megabit memory chips will be readily available at about $10 each, that is at less than $100 per megabyte. Finally we assume that VLSI design of special purpose chips will be readily carried out and that these too will be cheap (perhaps $10 each) when manufactured in quantity. On the other hand, logical designs involving many different VLSI chips with complex interconnections will remain expensive to design and construct. In other words, the cost/performance ratio will be minimized by using large numbers of each of very few chip types organized in as systematic and homogenous a design as possible.

Given this view, the question arises of how the organization of processors and memory is to be accomplished. One solution is to distribute the processors on some kind of network and to associate a portion of memory directly with each processor. This has engineering advantages and can be used very effectively when it can be ensured that the large majority of data treated by each processor is in fact in its associated memory unit. The rest of memory is then seen from each processor in a hierarchical way. The required data structure is difficult to guarantee in general, so that an architecture in which all memory is shared equally well (or badly) by all processors is likely to to be more easily programmed. Whether a truly

effective version of this shared memory architecture can be built is an open question. It is clear that the ideal version, in which each processor can access any memory location in one processor clock cycle, cannot be constructed because of limitations of local connectivity (i.e., of "fan in"). Schwartz [1] has given the name Paracomputer to this idealized architecture.

We contend that the current design of the NYU Ultracomputer is a feasible and effective approximation to a Paracomputer.

OVERVIEW

A simple idea of the organization is obtained by considering an array of individual processors (our nominal design calls for 4096), an equal number of memory units (containing, say, a megabyte each), and an intermediate switching network which permits any processor to reach any memory module. We propose to use an Omega network, [2] which is described below. Each processor will be capable of following its own instruction stream using distinct data. Such machines are often called "Multiple Instruction Stream- Multiple Data Stream (MIMD)".

Three components of secondary importance are also included: a memory-network interface (MNI), a local cache memory at each processor, and a processor-network interface (PNI). Besides the usual functions of reformatting information from or to the form required by the network, the MNI and PNI have certain novel features described below. The cache holds data local to the process which is active in the processor to which it is attached. As such it plays two roles: information can be accessed from the cache without the latency time of the network, and at the same time, the load on the network is decreased. It is worth emphasizing here that our design for the network results in a latency time under light loads of about 8 instruction times. This is moderately large, so that the use of a cache is worthwhile, but not so large as to preclude the effective use of shared data in the array of memory modules.

THE PROCESSORS; THE FETCH AND ADD INSTRUCTION

We assume each processor to be a standard microprocessor (of the future) with at least 32 bit arithmetic including fast floating-point operations. A novel instruction called "Fetch and Add" is also assumed. The effect of this instruction is as follows: the processor computes an increment e and an address a in shared memory. The operation

F&A(a,e)

returns the previous contents of a to the processor, and replaces the contents of a by its previous value increased by e. That is,

$$[a] \leftarrow [a] + e.$$

The operation of the Ultracomputer as a whole is supposed to be such that if two processors simultaneously issue an F&A for the same address a, then the outcome is as if they occured in some serial order. That is, if processor 1 calls for F&A(a,e_1) and processor 2 for F&A(a,e_2) and [a sub was originally A, then either

Processor 1 receives A and Processor 2 receives A + e_1
as though e_1 had arrived first.

or

Processor 1 receives A + e_2 and Processor 2 receives A
as though e_2 had arrived first.

In either case, [a] becomes A + e_1 + e_2 at the end. The case of three or more simultaneous F&A instructions directed at the same location works in an analogous way.

In fact the complete implementation of F&A will involve operations distributed among the PNIs, network switches, and MNIs. We remark here that were it possible to design a processor exclusively for use in an Ultracomputer, its instruction repertoire would include the F&A.[2]

Additional features of microprocessors that would enhance their suitability include the ability to issue a fetch from memory and continue executing instructions until the requested datum is actually needed. Local multiprogramming capability would be useful as an additional means of hiding network latency time.

THE NETWORK

The Omega network connecting 2^d inputs with 2^d outputs requires d stages of 2x2 switches. It has three properties that make it suitable for our purposes. Switching is very simple; at each stage a different bit of the memory module address is used to select which of the two possible output directions is to be taken. A second property is that its topology is a superposition of binary trees; each processor is the root of a tree whose 2^d leaves are the memory modules and each memory module is the root of a tree whose 2^d leaves are the processors. Finally, there is a unique path between a given processor and any memory module so that information directed simultaneously at a specific address by two processors must intersect at a switch.

---

[2]This is not absolutely essential; in our prototypes we plan to encode a F&A as a combination of existing instructions flagged by an otherwise unused address. This will be decoded by the PNI.

In attempting to build very large processor ensembles, it is essential that the maximum total throughput of the network (i.e., its "bandwidth") be proportional to the number of processors. To ensure this asymptotically, as the number of stages grows, two properties of the network are implied. One is that the network must be packet switched. That means that requests for data from memory are inserted into the network by the PNI as short messages which travel from switch to switch. In effect such messages are pipelined to memory and back.[3]

Since conflicts will occur, the second requirement on the network is that messages that cannot be transmitted because of conflicts be either combined with other messages, or stored at a switch for later transmission. That is, the switches must have some memory and some elementary processing power.

THE NETWORK SWITCHES

To support the requirement of large bandwidth, we have designed the network switches to be able to recognize messages aimed at the same address. If both are fetches, then only one fetch is transmitted toward memory but the identity of both processors is saved. When the fetched information returns to the switch in question, it is transmitted toward both processors. Conflicting stores are treated by suppressing one of them.

The treatment required for concurrent F&A instructions targeted at the same address is interesting. Suppose a switch receives F&A(a,e_1) and F&A(a,e_2) simultaneously. It will be programmed to save e_1, and transmit a new message, namely F&A(a,$e_1 + e_2$). When the response returns with the previous contents of a, it is transmitted back toward one processor and [a] + e_1 is transmitted toward the other.

Precisely because a binary tree exists between all processors and any memory module, when any number of F&A are issued simultaneously for the same a, they can combine in pairs at successive stages of the network while converging toward a, and split into different results on the way back. Thus any number of F&A's for the same address may be carried out in the same time as a single one. As we shall see, this property will be important for the coordination of very many closely coupled processes.

As indicated above, when messages arrive at a switch for different addresses, one is transmitted, one saved for later. The required memory is designed as a shifting array on a chip (i.e., as a "systolic" queue) in such a way that identical addresses can be identified associatively.

---

[3]The alternative, (a "circuit switched" network) in which a path is cleared from processor to memory and kept open while information flows both ways, will suffer from an increasing probability of interference as the network increases in size.

THE MEMORY NETWORK INTERFACE

The MNI carries out the final addition associated with an F&A(a,e). That is, the MNI retrieves [a] and transmits it back toward the requesting processor, adds e, and stores the sum in a.

A LITTLE ABOUT SOFTWARE

OVERVIEW

It is impossible to include more than a sketch of the aims and accomplishments of the software research for Ultracomputer systems.

From the beginning of our research, attention has been paid to the question of how friendly the machine would prove for the ordinary scientific programmer. It is considered important that radically new languages, programming style, or user interfaces be unnecessary, and that it prove possible to transport existing major software without complete revision.

A property of an operating system which is considered highly desirable for machines of this class is the ability to handle simultaneously jobs of all sizes for many users. This would provide rapid turnaround for small jobs so necessary during program development. In addition, as we shall see below, a full complement of processors can only be used efficiently on very large problems. Economic use of such a machine requires that a subset of the available processors be readily available for work on a small or medium scale.

SIMULATORS, LANGUAGES, AND PROGRAMMING STYLE

A very early development in our research was a simulator called "WASHCLOTH" [3] in which a CDC 6600 was programmed to act like a paracomputer with 6600's as the individual processors. This was accomplished by interpreting each instruction serially, with pointers to the private memory for each processor. In this way, existing FORTRAN compilers could be used without modification. Fetch and Add was implemented as a subroutine call. Shared memory was assumed to be confined to unlabeled COMMON.

Naturally one expects that FORTRAN will have to be extended to provide the flexibility needed for an Ultracomputer. As indicated, a distinction between private and shared memory is needed. Probably still more general schemes for sharing variables with subprocesses will be used as well. In addition, commands to create parallel subprocesses, to wait for their completion, to carry out iterations entirely in parallel, and so on will have to be specified and included in compilers. A general compiler would also

include direct access to the Fetch and Add, permitting a programmer to coordinate processes more tightly than would be possible at a higher level.

It is likely that a more advanced language will emerge which will prove more natural and more expressive for controlling parallel asynchronous processes. It is my belief that this will be the product of an extended period of experiment and experience with scientific programming using lower level languages.

ELEMENTS OF AN OPERATING SYSTEM

The requirement for an operating system in which jobs can spawn cooperating tasks and for one which can support multiprogramming are not very different, at least at a basic level. For a machine like the Ultracomputer in which many thousands of tasks may be active or awaiting the attention of a processor, it will be very important that there be no centralized monitor or operating system kernel residing in a single processor, because there would be serious contention for that processor. The result would be a bottleneck at the point of creating new tasks or assigning them to idle processors. Our approach to this problem has been to assume that each processor would carry out the functions of the operating system when required. That is, when idle, a processor would access a global queue of tasks awaiting attention. Similarly any processor called upon to spawn subtasks or create entirely new jobs would make enter appropriate items in the same global queue. On the face of it, this only moves the difficulty of bottlenecks elsewhere, since the standard way of entering and deleting from queues requires that access to them be blocked (i.e., by a "critical section") to prevent more than one processor from either making a queue entry in the same slot or withdrawing the same information erroneously.

The properties of the Fetch and Add make it ideal for modifying queue pointers; that is, if F&A(p,1) is used by two processors to increment a queue pointer stored in p, then it is guaranteed that they will obtain different values as required for correct operation. The properties of both network and switches guarantee that concurrent modifications of queue pointers take place without delay.

The Fetch and Add can be used to implement all necessary coordination primitives for an operating system, including semaphores, handling of readers-writers, and so forth.

STUDY OF APPLICATIONS UNDER SIMULATION

OVERVIEW

We have carried out studies in which a number of programs written in FORTRAN were executed under control of an interpreter which made a

serial computer act like a parallel computer. The simulator accumulated statistics from which one could infer the effectiveness of a particular program organization and related data. In particular, the total number of instructions, both useful and idle, was counted. The number of references to private and thus cacheable variables relative to references to shared data was also scored to help estimate the traffic on the network. the traffic on the network.

If one considers only the narrow (but very important) issue of the efficiency of processor utilization, then only the total number of operations matters. From now on we shall assume that N processors are dedicated to a given problem so that if any of them must wait for the completion of work by other processors, it can only idle. Since, in principle, such processors could in fact carry out useful work for other jobs, we will get a conservative estimate of efficiency.

If the program in question could be divided among the N processors so that each carried out exactly 1/N of the work, the job would be completed N times as fast as on a single processor. Normally this is not possible; some overhead associated with process creation, with synchronization, and with idling will occur. Let C(N) be the average number of instructions per processor when N are used. The efficiency is the ratio of the instructions per processor for the ideal parallel algorithm, C(1)/N, to C(N). That is,

$$E = \frac{C(1)}{NC(N)}$$

It is not surprising that if $C(1) << N$, it is impossible to find an efficient parallel decomposition, since at least N - C(1) processors will have nothing to do, to say nothing of the organizational overhead required. In other words, a problem must be large enough to begin with to warrant the use of N processors. There is, of course, no lack of such problems, but it is clearly important to understand this point in extrapolating the use of very large parallel machines for problems of the size which can generally be solved today.

We have simulated the use of large parallel arrays in a variety of fields from transport Monte Carlo to incompressible fluid flow to matrix transformation algorithms. Our studies have required both writing new programs and adapting existing codes, some of the latter poorly written and documented. In every case, the difference from writing FORTRAN on a sequential machine required fairly simple, usually high-level reorganization which differed from program to program and which had to be recognized in advance by the programmer. In every case, a parallel program was created which was efficient for computing on a sufficiently large scale. Thus, while 256 processors cannot be used very effectively to transform a 32x32 matrix to tridiagonal form, the efficiency being only 25%, they can be used with 96% efficiency for a 256x256 matrix. In the same way large problems of three-

dimensional fluid flow were found to be efficiently handled by 4096 processors.

TRANSPORT MONTE CARLO

Monte Carlo codes of the type used in reactor and shielding calculations would seem very well suited to treatment by any parallel machine of the type we have been discussing. After all, what could be needed except the ability to follow independently and asynchronously many histories in parallel? A little reflection shows that not all possible MIMD architectures will be well suited to large Monte Carlo calculations. In particular, if we assume that it is important to treat large cross sections sets, large geometrical descriptions, and to obtain a large set of fluxes and their errors, then the limitations of hierarchical memories become obvious.

The advantages of the Ultracomputer organization are several. If, as we plan, it will be possible to flag cross section and geometrical data as cacheable without restriction (i.e., as "read only"), then most of the references to shared data can be effectively treated.

Updating particle tallies in a large number of different spatial, energy, time, and direction bins by different processors presents a serious problem in some machines. This difficulty is well resolved by the Fetch and Add instruction. The computation of statistical error remains as a problem at least when the number of bins is large.[4] If one settles for errors computed from large numbers of histories taken together as a statistical batch, then the problem is easily solved by waiting for all histories in a batch to complete before proceeding. The main departure from an efficiency of 100% then comes from the straggling in the time for completion of each batch. The effect of straggling is much reduced by assigning new histories to processors as they become idle,[5] and can be made very small by using two or more replications of the memory space reserved for recording the tallies.

These issues and others have been considered by Zhong and Kalos [4] for fixed source and eigenvalue calculations. They used simulations, algorithmic analysis of the simulations, and analytic treatment of approximate stochastic models of the efficiency. Their conclusion is that for large problems of this class, the efficiency becomes greater than 90%.

---

[4] i.e., so large that all bins for all statistical groups can not be kept in memory.

[5] Rather than preassigning a fixed fraction of histories to a processor.

CONCLUSIONS

At the present time we see no reason why a machine like the Ultracomputer could not be built economically and used effectively by scientific programmers. For the very large problems not being attacked now for lack of computing resources, efficiencies of close to 100% should be attainable. It is interesting to note that on vector machines good utilization of the peak computing rate is about 15%.

What remains uncertain now is whether and how fast one can proceed with more stringent examination of the practicability of such a machine. It seems clear that a fairly large model or prototype needs to be built.

REFERENCES

(1) J. T. Schwartz, "Ultracomputers," *ACM TOPLAS*, (1980) p. 484.

(2) A. Gottlieb and J.T. Schwartz, "Networks and Algorithms for Very Large Scale Parallel Computations," *Computer*, January 1982.

(3) Allan Gottlieb, "WASHCLOTH - The Logical Successor to Soapsuds," Ultracomputer Note #12, Courant Institute, NYU, 1980.

(4) Y-Q Zhong and M. H. Kalos, "Monte Carlo Transport Calculations on an Ultracomputer," Ultracomputer Note #46, Courant Institute, NYU, 1982.